Virtual processors and threads

These topics describe virtual processors

Explain how threads run within the virtual processors

Explain how the database server uses virtual processors and threads to improve performance.

Virtual processors

A virtual processor is a process that the operating system schedules for processing.

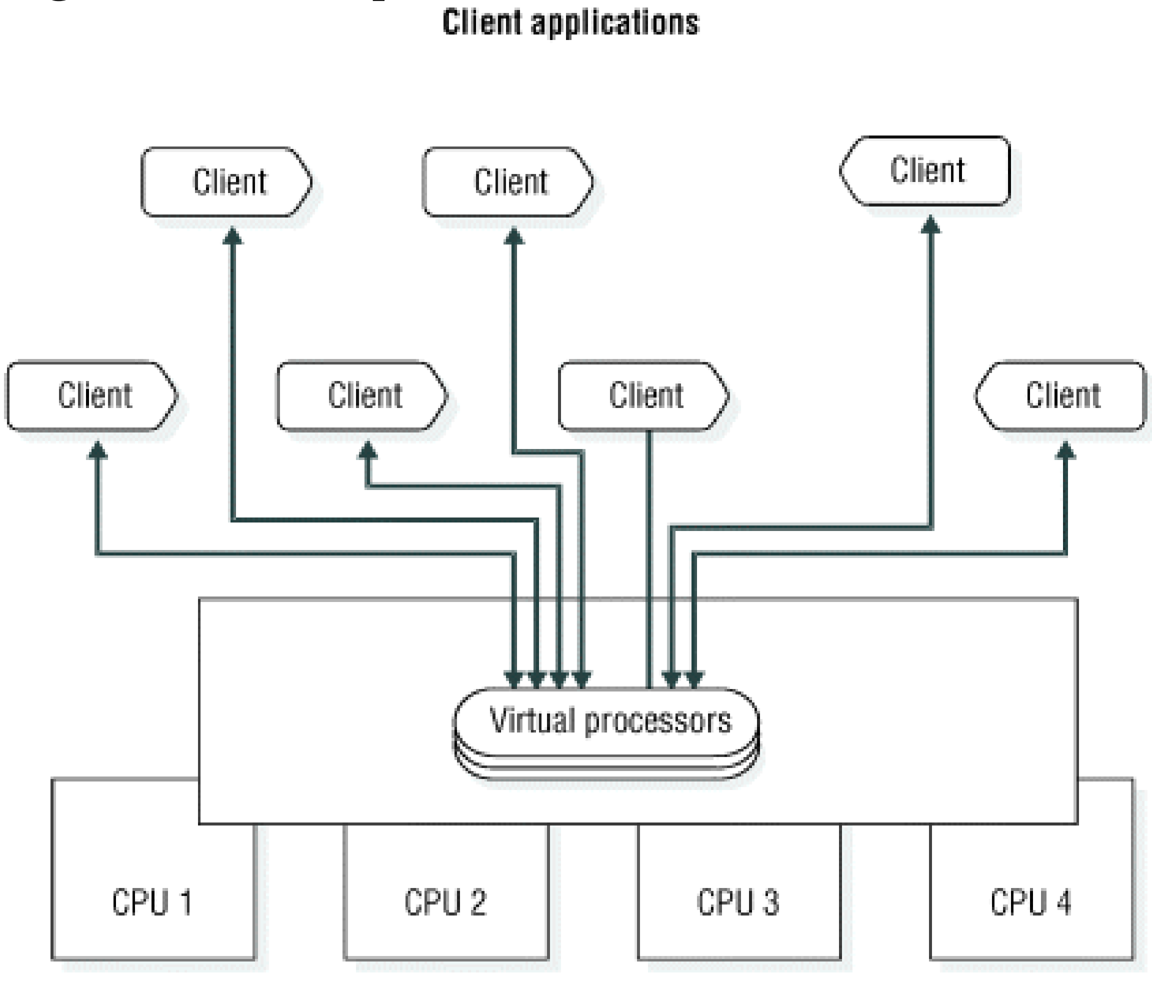
Database server processes are called *virtual processors* because the way they function is similar to the way that a CPU functions in a computer.

Just as a CPU runs multiple operating-system processes to service multiple users, a database server virtual processor runs multiple threads to service multiple SQL client applications.

The following figure illustrates the relationship of client applications to virtual processors.

A small number of virtual processors serve a much larger number of client applications or queries.

Figure 1. Virtual processors



Threads

A thread is a task for a virtual processor in the same way that the virtual processor is a task for the CPU.

The virtual processor is a task that the operating system schedules for execution on the CPU.

A database server thread is a task that the virtual processor schedules internally for processing.

Threads are sometimes called *lightweight processes* because they are like processes, but they make fewer demands on the operating system.

Database server virtual processors are multithreaded because they run multiple concurrent threads.

The nature of threads is as follows.

Operating system	Action
UNIX	A thread is a task that the virtual processor schedules internally for processing.
Windows	A thread is a task that the virtual processor schedules internally for processing. Because the virtual processor is implemented as a Windows thread, database server threads run within Windows threads.

A virtual processor runs threads on behalf of SQL client applications (*session* threads) and also to satisfy internal requirements (*internal* threads).

In most cases, for each connection by a client application, the database server runs one session thread.

The database server runs internal threads to accomplish, among other things, database I/O, logging I/O, page cleaning, and administrative tasks.

A *user thread* is a database server thread that services requests from client applications.

User threads include session threads, called **sqlexec** threads, which are the primary threads that the database server runs to service client applications.

User threads also include a thread to service requests from the **`onmode`** utility, threads for recovery, B-tree scanner threads, and page-cleaner threads.

To display active user threads, use **`onstat -u`**.

Advantages of virtual processors

A virtual processor provides a number of advantages.

Compared to a database server process that services a single client application, the dynamic, multithreaded nature of a database server virtual processor provides the following advantages:

- Virtual processors can share processing.
- Virtual processors save memory and resources.
- Virtual processors can perform parallel processing.
- You can start additional virtual processors and terminate active CPU virtual processors while the database server is running.
- You can bind virtual processors to CPUs.

Shared processing

Virtual processors in the same class have identical code and share access to both data and processing queues in memory.

Any virtual processor in a class can run any thread that belongs to that class.

Generally, the database server tries to keep a thread running on the same virtual processor because moving it to a different virtual processor can require some data from the memory of the processor to be transferred on the bus.

When a thread is waiting to run, however, the database server can migrate the thread to another virtual processor because the benefit of balancing the processing load outweighs the amount of overhead incurred in transferring the data.

Shared processing within a class of virtual processors occurs automatically and is transparent to the database user.

Save memory and resources

The database server is able to service many clients with a small number of server processes compared to architectures that have one client process to one server process by running a thread, rather than a process, for each client.

Multithreading permits more efficient use of the operating-system resources because threads share the resources allocated to the virtual processor.

All threads that a virtual processor runs have the same access to the virtual-processor memory, communication ports, and files.

The virtual processor coordinates access to resources by the threads.

Individual processes, though, each have a distinct set of resources, and when multiple processes require access to the same resources, the operating system must coordinate the access.

Generally, a virtual processor can switch from one thread to another faster than the operating system can switch from one process to another.

When the operating system switches between processes, it must stop one process from running on the processor, save its current processing state (or context), and start another process.

Both processes must enter and exit the operating-system kernel, and the contents of portions of physical memory might require replacement.

Threads, though, share the same virtual memory and file descriptors.

When a virtual processor switches from one thread to another, the switch is from one path of execution to another.

The virtual processor, which is a process, continues to run on the CPU without interruption.

Parallel processing

When a client initiates index building, sorting, or logical recovery, the database server creates multiple threads to work on the task in parallel, using as much of the computer resources as possible.

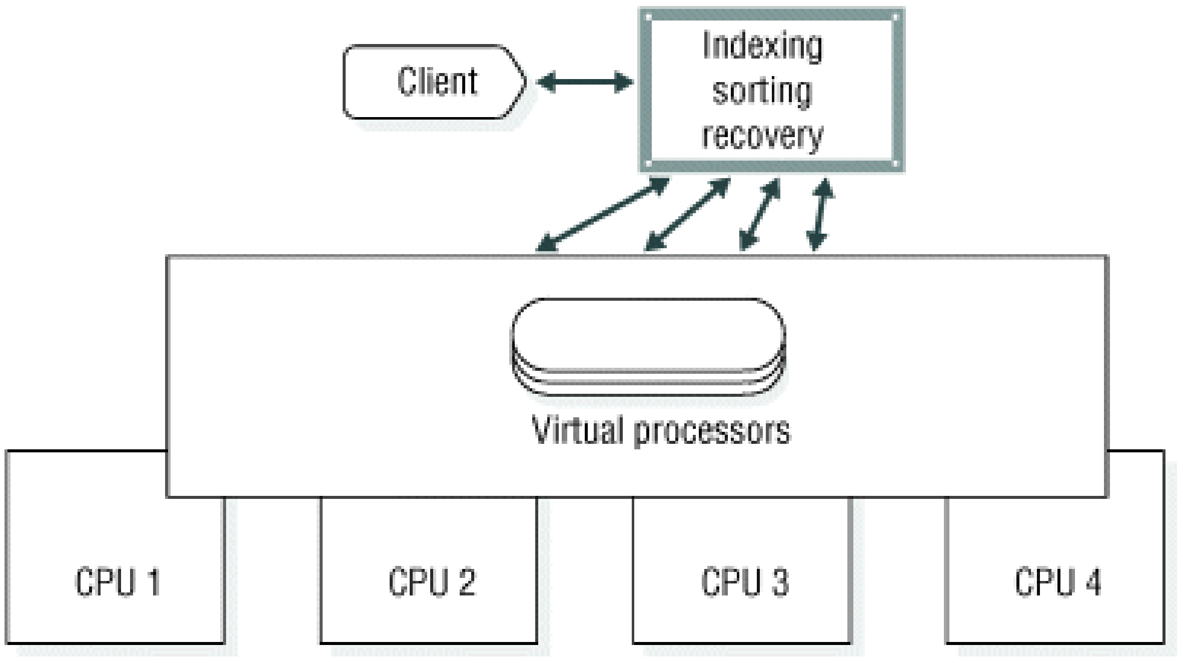
While one thread is waiting for I/O, another can be working.

In the following cases, virtual processors of the CPU class can run multiple session threads, working in parallel, for a single client:

- Index building
- Sorting
- Recovery
- Scanning
- Joining
- Aggregation
- Grouping
- User-defined-routine (UDR) execution

The following figure illustrates parallel processing.

Figure 1. Parallel processing



Add and drop virtual processors in online mode

You can add virtual processors to meet increasing demands for service while the database server is running.

If the virtual processors of a class become compute bound or I/O bound (meaning that CPU work or I/O requests are accumulating faster than the current number of virtual processors can process them), you can start additional virtual processors for that class to distribute the processing load further.

Windows only: In Windows, you cannot drop a virtual processor of any class.

While the database server is running, you can drop virtual processors of the CPU or a user-defined class.

Bind virtual processors to CPUs

You can use some multiprocessor systems to bind a process to a particular CPU. This feature is called *processor affinity*.

On multiprocessor computers for which the database server supports processor affinity, you can bind CPU virtual processors to specific CPUs in the computer.

When you bind a CPU virtual processor to a CPU, the virtual processor runs exclusively on that CPU.

This operation improves the performance of the virtual processor because it reduces the amount of switching between processes that the operating system must do.

Binding CPU virtual processors to specific CPUs also enables you to isolate database work on specific processors on the computer, leaving the remaining processors free for other work.

Only CPU virtual processors can be bound to CPUs.

How virtual processors service threads

A virtual processor services multiple threads concurrently by switching between them.

At a given time, a virtual processor can run only one thread.

A virtual processor runs a thread until it yields.

When a thread yields, the virtual processor switches to the next thread that is ready to run.

The virtual processor continues this process, eventually returning to the original thread when that thread is ready to continue.

Some threads complete their work, and the virtual processor starts new threads to process new work.

Because a virtual processor continually switches between threads, it can keep the CPU processing continually.

The speed at which processing occurs produces the appearance that the virtual processor processes multiple tasks simultaneously and, in effect, it does.

Running multiple concurrent threads requires scheduling and synchronization to prevent one thread from interfering with the work of another.

Virtual processors use the following structures and methods to coordinate concurrent processing by multiple threads:

- Control structures
- Context switching
- Stacks
- Queues
- Mutexes

Control structures

When a client connects to the database server, the database server creates a *session* structure, which is called a *session control block*, to hold information about the connection and the user.

A session begins when a client connects to the database server, and it ends when the connection terminates.

Next, the database server creates a thread structure, which is called a *thread-control block* (TCB) for the session, and initiates a primary thread (**sqlexec**) to process the client request.

When a thread *yields*—that is, when it pauses and allows another thread to run—the virtual processor saves information about the state of the thread in the thread-control block.

This information includes the content of the process system registers, the program counter (address of the next instruction to execute), and the stack pointer.

This information constitutes the context of the thread.

In most cases, the database server runs one primary thread per session.

In cases where it performs parallel processing, however, it creates multiple session threads for a single client, and, likewise, multiple corresponding thread-control blocks.

Context switching

A virtual processor switches from running one thread to running another one by *context switching*.

The database server does not preempt a running thread, as the operating system does to a process, when a fixed amount of time (time-slice) expires.

Instead, a thread yields at one of the following points:
- A predetermined point in the code
- When the thread can no longer execute until some condition is met

When the amount of processing required to complete a task would cause other threads to wait for an undue length of time, a thread yields at a predetermined point.

The code for such long-running tasks includes calls to the yield function at strategic points in the processing.

When a thread performs one of these tasks, it yields when it encounters a yield function call.

Other threads in the ready queue then get a chance to run.

When the original thread next gets a turn, it resumes executing code at the point immediately after the call to the yield function.

Predetermined calls to the yield function allow the database server to interrupt threads at points that are most advantageous for performance.

A thread also yields when it can no longer continue its task until some condition occurs.

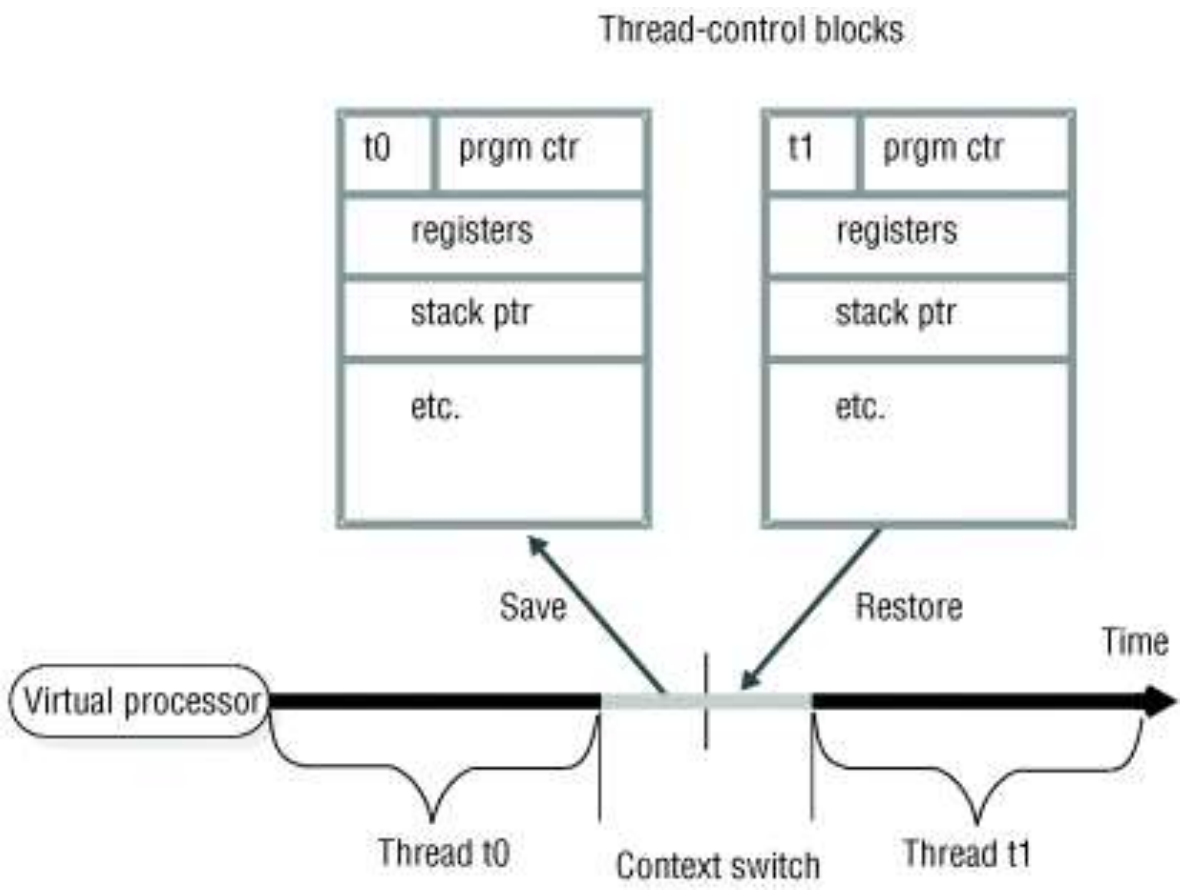
For example, a thread yields when it is waiting for disk I/O to complete, when it is waiting for data from the client, or when it is waiting for a lock or other resource.

When a thread yields, the virtual processor saves its context in the thread-control block.

Then the virtual processor selects a new thread to run from a queue of ready threads, loads the context of the new thread from its thread-control block, and begins executing at the new address in the program counter.

The following figure illustrates how a virtual processor accomplishes a context switch.

Figure 1. Context switch: how a virtual processor switches from one thread to another



Stacks

The database server allocates an area in the virtual portion of shared memory to store nonshared data for the functions that a thread executes.

This area is called the *stack*.

The stack enables a virtual processor to protect the nonshared data of a thread from being overwritten by other threads that concurrently execute the same code.

For example, if several client applications concurrently perform SELECT statements, the session threads for each client execute many of the same functions in the code.

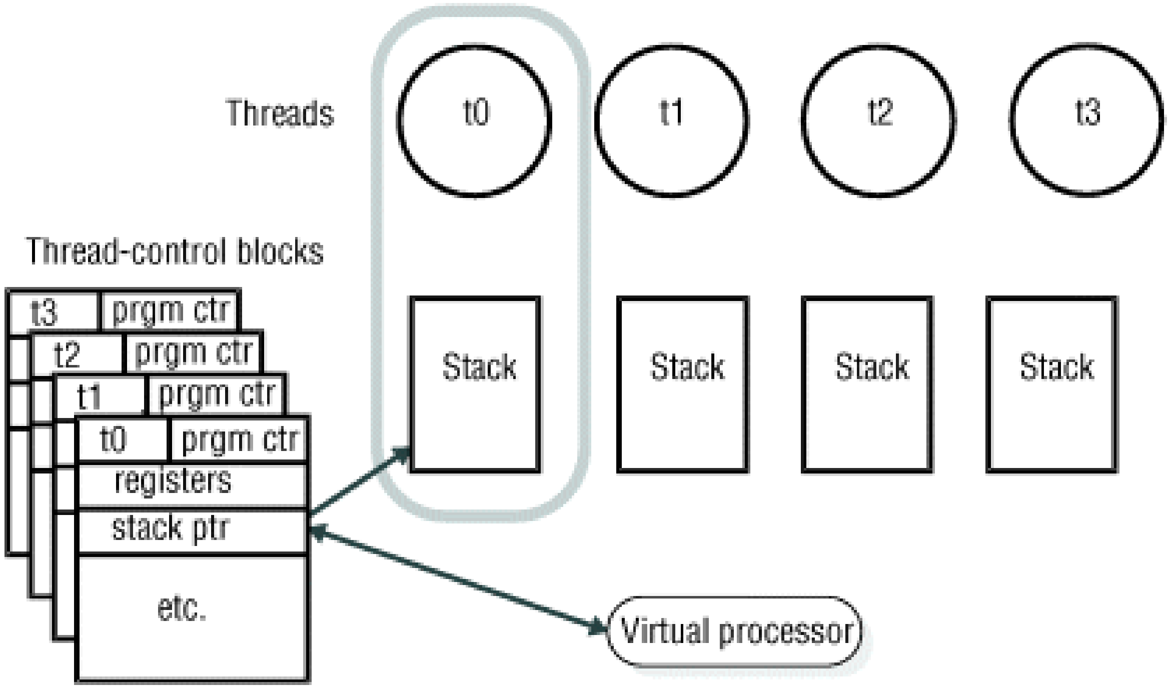
If a thread did not have a private stack, one thread might overwrite local data that belongs to another thread within a function.

When a virtual processor switches to a new thread, it loads a stack pointer for that thread from a field in the thread-control block.

The stack pointer stores the beginning address of the stack. The virtual processor can then specify offsets to the beginning address to access data within the stack.

The figure illustrates how a virtual processor uses the stack to segregate nonshared data for session threads.

Figure 1. Virtual processors segregate nonshared data for each user



Queues

The database server uses three types of queues to schedule the processing of multiple, concurrently running threads.

Virtual processors of the same class share queues.

This fact, in part, enables a thread to migrate from one virtual processor in a class to another when necessary.

Ready queues

Ready queues hold threads that are ready to run when the current (running) thread yields.

When a thread yields, the virtual processor picks the next thread with the appropriate priority from the ready queue.

Within the queue, the virtual processor processes threads that have the same priority on a first-in-first-out (FIFO) basis.

On a multiprocessor computer, if you notice that threads are accumulating in the ready queue for a class of virtual processors (indicating that work is accumulating faster than the virtual processor can process it), you can start additional virtual processors of that class to distribute the processing load.

Sleep queues

Sleep queues hold the contexts of threads that have no work to do at a particular time.

A thread is put to sleep either for a specified period of time or forever.

The administration class (ADM) of virtual processors runs the system timer and special utility threads.

Virtual processors in this class are created and run automatically.

No configuration parameters affect this class of virtual processors.

The ADM virtual processor wakes up threads that have slept for the specified time.

A thread that runs in the ADM virtual processor checks on sleeping threads at one-second intervals.

If a sleeping thread has slept for its specified time, the ADM virtual processor moves it into the appropriate ready queue.

A thread that is sleeping for a specified time can also be explicitly awakened by another thread.

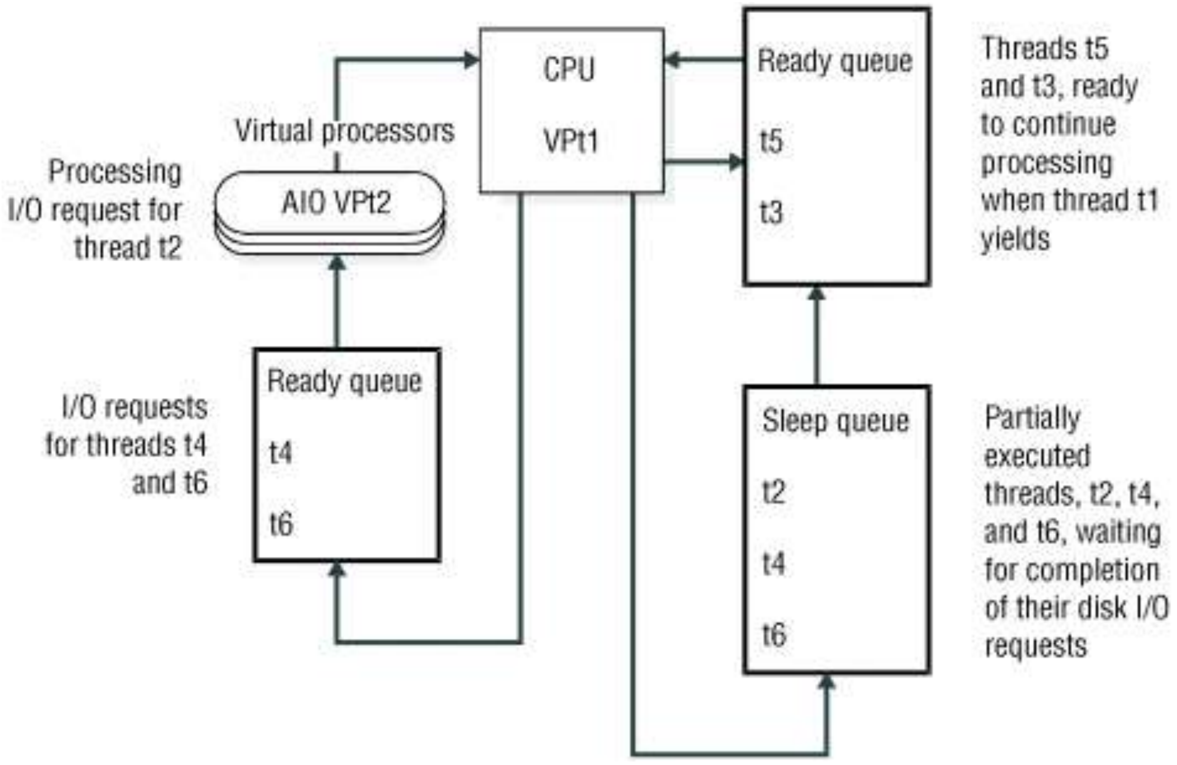
A thread that is sleeping forever is awakened when it has more work to do.

For example, when a thread that is running on a CPU virtual processor must access a disk, it issues an I/O request, places itself in a sleep queue for the CPU virtual processor, and yields.

When the I/O thread notifies the CPU virtual processor that the I/O is complete, the CPU virtual processor schedules the original thread to continue processing by moving it from the sleep queue to a ready queue.

The following figure illustrates how the database server threads are queued to perform database I/O.

Figure 1. How database server threads are queued to perform database I/O



CPU
VPt1
Virtual processors
AIO VPt2
Processing I/O request for thread t2
Ready queue
t5
t3
Threads t5 and t3, ready to continue processing when thread t1 yields
Ready queue
t4
t6
I/O requests for threads t4 and t6
Sleep queue
t2
t4
t6
Partially executed threads, t2, t4, and t6, waiting for completion of their disk I/O requests

Wait queues

Wait queues hold threads that must wait for a particular event before they can continue to run.

Wait queues coordinate access to shared data by threads.

When a user thread tries to acquire the logical-log latch but finds that the latch is held by another user, the thread that was denied access puts itself in the logical-log wait queue.

When the thread that owns the lock is ready to release the latch, it checks for waiting threads, and, if threads are waiting, it wakes up the next thread in the wait queue.

Mutexes

A mutex (*mut*ually *ex*clusive), also called a *latch*, is a latching mechanism that the database server uses to synchronize access by multiple threads to shared resources.

Mutexes are similar to semaphores, which some operating systems use to regulate access to shared data by multiple processes.

However, mutexes permit a greater degree of parallelism than semaphores.

A mutex is a variable that is associated with a shared resource such as a buffer.

A thread must acquire the mutex for a resource before it can access the resource.

Other threads are excluded from accessing the resource until the owner releases it.

A thread acquires a mutex, after a mutex becomes available, by setting it to an in-use state.

The synchronization that mutexes provide ensures that only one thread at a time writes to an area of shared memory.

Virtual processor classes

Each class of virtual processor is dedicated to processing certain types of threads.

The following table shows the classes of virtual processors and the types of processing that they do.

The number of virtual processors of each class that you configure depends on the availability of physical processors (CPUs), hardware memory, and the database applications in use.

Virtual-processor class	**Category**	**Purpose**
ADM	Administrative	Performs administrative functions.
ADT	Auditing	Performs auditing functions.
AIO	Disk I/O	Performs nonlogging disk I/O. If KAIO is used, AIO virtual processors perform I/O to cooked disk spaces.
BTS	Basic text searching	Runs basic text search index operations and queries.
CPU	Central processing	Runs all session threads and some system threads. Runs thread for kernel asynchronous I/O (KAIO) where available. Can run a single poll thread, depending on configuration.
CSM	Communications Support Module	Performs communications support service operations. **Note:** Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security

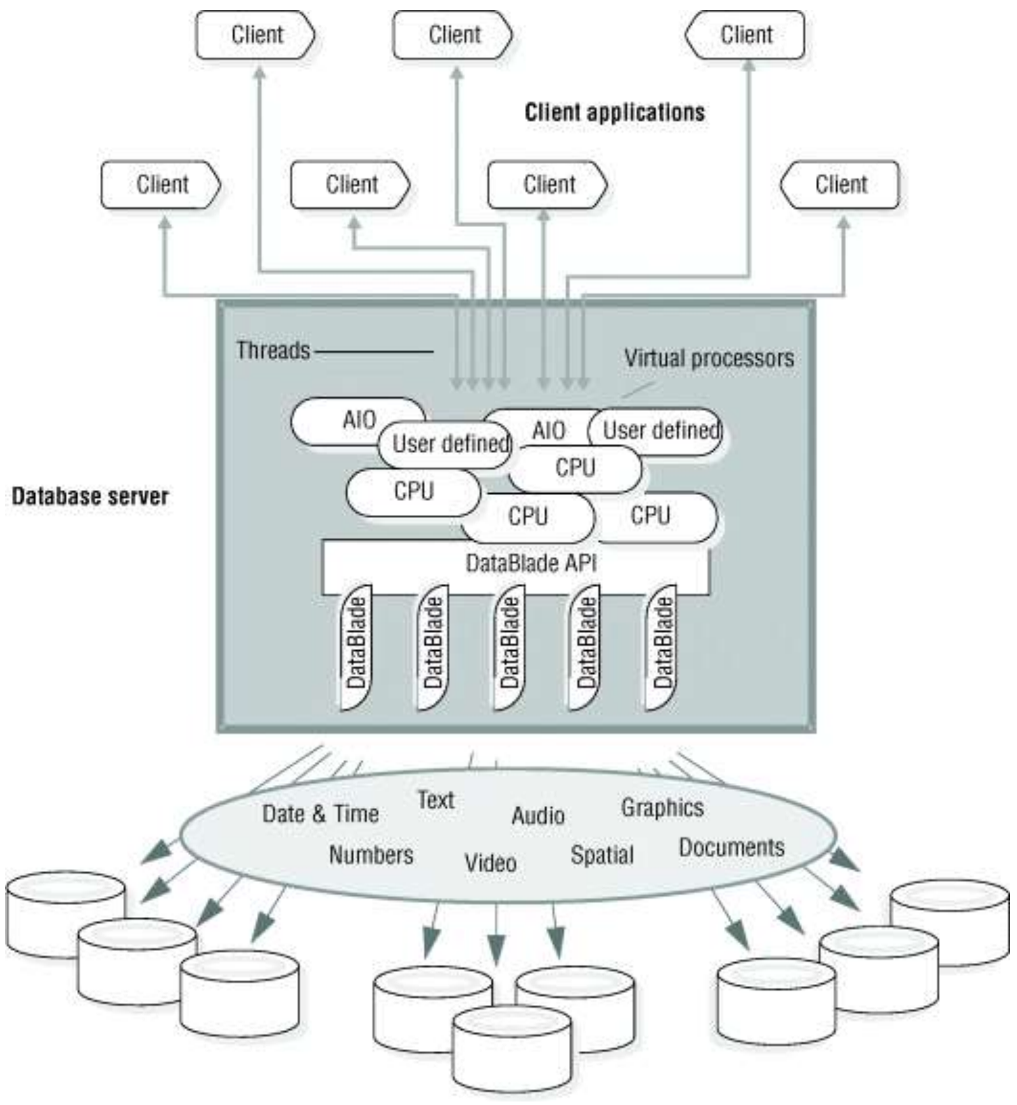
Virtual-processor class	Category	Purpose
		(TLS)/Secure Sockets Layer (SSL) instead.
dwavp	Data warehousing	Runs the administrative functions and procedures for Informix® Warehouse Accelerator on a database server that is connected to Informix Warehouse Accelerator.
Encrypt	Encryption	Used by the database server when encryption or decryption functions are called. On Windows systems, the number of **`encrypt`** virtual processors is always set to 1, regardless of the value that is set in the `onconfig` file.
IDSXMLVP	XML publishing	Runs XML publishing functions.

Virtual-processor class	**Category**	**Purpose**
JVP	Java™ UDR	Runs Java UDRs. Contains the Java Virtual Machine (JVM).
LIO	Disk I/O	Writes to the logical-log files (internal class) if they are in cooked disk space.
MQ	MQ messaging	Performs MQ messaging transactions.
MSC	Miscellaneous	Services requests for system calls that require a very large stack.
PIO	Disk I/O	Writes to the physical-log file (internal class) if it is in cooked disk space.
SHM	Network	Performs shared memory communication.
SOC	Network	Uses sockets to perform network communication.
`tenant`	Multitenancy	Runs session threads for tenant databases. Tenant virtual processors are a special case of user-defined

Virtual-processor class	**Category**	**Purpose**
		processors that are specific to tenant databases.
TLI	Network	Uses the Transport Layer Interface (TLI) to perform network communication.
WFSVP	Web feature service	Runs web feature service routines.
`classname`	User defined	Runs user-defined routines in a thread-safe manner so that if the routine fails, the database server is unaffected.

Table 1. Virtual-processor classes

The following figure illustrates the major components and the extensibility of the database server.Figure 1. Database server



CPU virtual processors

The CPU virtual processor runs all session threads (the threads that process requests from SQL client applications) and some internal threads.

Internal threads perform services that are internal to the database server.

For example, a thread that listens for connection requests from client applications is an internal thread.

Each CPU virtual processor can have a private memory cache associated with it.

Each private memory cache block consists of 1 to 32 memory pages, where each memory page is 4096 bytes.

The database server uses the private memory cache to improve access time to memory blocks.

Use the VP_MEMORY_CACHE_KB configuration parameter to enable a private memory cache and specify information about the memory cache.

User-defined classes of virtual processors

You can define special classes of virtual processors to run user-defined routines or to run a DataBlade module.

User-defined routines are typically written to support user-defined data types.

If you do not want a user-defined routine to run in the CPU class, which is the default, you can assign it to a user-defined class of virtual processors (VPs).

User-defined classes of virtual processors are also called *extension virtual processors*.

Determine the number of user-defined virtual processors needed

You can specify as many user-defined virtual processors as your operating system allows.

If you run many UDRs or parallel PDQ queries with UDRs, you must configure more user-defined virtual processors.

User-defined virtual processors

User-defined classes of virtual processors protect the database server from *ill-behaved* user-defined routines.

An ill-behaved user-defined routine has at least one of the following characteristics:

- Does not yield control to other threads
- Makes blocking operating-system calls
- Modifies the global VP state

A well-behaved C-language UDR has none of these characteristics. Run only well-behaved C-language UDRs in a CPU VP.

Warning: Execution of an ill-behaved routine in a CPU VP can cause serious interference with the operation of the database server, possibly causing it to fail or behave erratically. In addition, the routine itself might not produce correct results.

To ensure safe execution, assign any ill-behaved user-defined routines to a user-defined class of virtual processors.

User-defined VPs remove the following programming restrictions on the CPU VP class:

- The requirement to yield the processor regularly
- The requirement to eliminate blocking I/O calls

Functions that run in a user-defined virtual-processor class are not required to yield the processor, and they might issue direct file-system calls that block further processing by the virtual processor until the I/O is complete.

The normal processing of user queries is not affected by ill-behaved traits of a C-language UDR because these UDRs do not execute in CPU virtual processors.

Specify user-defined virtual processors

The VPCLASS parameter with the *vpclass* option defines a user-defined VP class. You also can specify a nonyielding user-defined virtual processor.

Assign a UDR to a user-defined virtual-processor class

The SQL CREATE FUNCTION statement registers a user-defined routine.

The following CREATE FUNCTION statement registers the user-defined routine, **`GreaterThanEqual()`**, and specifies that calls to this routine are executed by the user-defined VP class named UDR:

```
CREATE FUNCTION
GreaterThanEqual(ScottishName,
ScottishName)
   RETURNS boolean
   WITH (CLASS = UDR  )
   EXTERNAL NAME
'/usr/lib/objects/udrs.so'
   LANGUAGE C
```

To execute this function, the `onconfig` file must include a VPCLASS parameter that defines the UDR class.

If not, calls to the **`GreaterThanEqual`** function fail.

The CLASS routine modifier can specify any name for the VP class.

This class name is not required to exist when you register the UDR. However, when you try to run a UDR that specifies a user-defined VP class for its execution, this class must exist and have virtual processors assigned to it.

To configure the UDR class, include a line similar to the following one in the `onconfig` file.

This line configures the UDR class with two virtual processors and with no priority aging.

```
VPCLASS       UDR   ,num=2,noage
```

The preceding line defines the UDR VP class as a yielding VP class; that is, this VP class allows the C-language UDR to yield to other threads that must access to the UDR VP class.

Add and drop user-defined virtual processors in online mode

You can add or drop virtual processors in a user-defined class while the database server is online.

Tenant virtual processor class

Tenant virtual processor classes are specific to tenant databases.

If you configure multitenancy for your Informix® instance, you can specify that session threads for tenant databases are run in tenant virtual processors instead of CPU virtual processors.

You can create a tenant virtual processor class by defining the class and the number of virtual processors when you create a tenant database.

You can assign a tenant virtual processor class to multiple tenant databases.

Set the VP_MEMORY_CACHE_KB configuration parameter to create a private memory cache for each CPU virtual processor and tenant virtual processor.

A tenant virtual processor class is automatically dropped when all tenant databases that include the virtual processor class in their definitions are dropped.

Java virtual processors

Java™ UDRs and Java applications run on specialized virtual processors, called *Java virtual processors* (JVPs).

A JVP embeds a Java virtual machine (JVM) in its code.

A JVP has the same capabilities as a CPU VP in that it can process complete SQL queries.

You can specify as many JVPs as your operating system allows.

If you run many Java UDRs or parallel PDQ queries with Java UDRs, you must configure more JVPs.

Use the VPCLASS configuration parameter with the `jvp` keyword to configure JVPs.

Disk I/O virtual processors

The following classes of virtual processors perform disk I/O: PIO (physical-log I/O), LIO (logical-log I/O), AIO (asynchronous I/O), and CPU (kernel-asynchronous I/O).

The PIO class performs all I/O to the physical-log file, and the LIO class performs all I/O to the logical-log files, *unless* those files are in raw disk space and the database server has implemented KAIO.

On operating systems that do not support KAIO, the database server uses the AIO class of virtual processors to perform database I/O that is not related to physical or logical logging.

The database server uses the CPU class to perform KAIO when it is available on a platform.

If the database server implements KAIO, a KAIO thread performs all I/O to raw disk space, including I/O to the physical and logical logs.

UNIX only: To find out if your UNIX platform supports KAIO, see the machine notes file.

Windows only: Windows supports KAIO.

I/O priorities

The database server prioritizes disk I/O by assigning different types of I/O to different classes of virtual processors and by assigning priorities to the nonlogging I/O queues.

Prioritizing ensures that a high-priority log I/O, for example, is never queued behind a write to a temporary file, which has a low priority.

The database server prioritizes the different types of disk I/O that it performs, as the table shows.

Priority	**Type of I/O**	**VP class**
1st	Logical-log I/O	CPU or LIO
2nd	Physical-log I/O	CPU or PIO
3rd	Database I/O	CPU or AIO
3rd	Page-cleaning I/O	CPU or AIO
3rd	Read-ahead I/O	CPU or AIO

Table 1. How database server prioritizes disk I/O

Logical-log I/O

The LIO class of virtual processors performs I/O to the logical-log files.

I/O is performed to logical-log files in the following cases:

- KAIO is not implemented.
- The logical-log files are in cooked disk space.

Only when KAIO is implemented and the logical-log files are in raw disk space does the database server use a KAIO thread in the CPU virtual processor to perform I/O to the logical log.

The logical-log files store the data that enables the database server to roll back transactions and recover from system failures.

I/O to the logical-log files is the highest priority disk I/O that the database server performs.

If the logical-log files are in a dbspace that
is **not** mirrored, the database server runs only one LIO virtual processor.

If the logical-log files are in a dbspace that is mirrored, the database server runs two LIO virtual processors.

This class of virtual processors has no parameters associated with it.

Physical-log I/O

The PIO class of virtual processors performs I/O to the physical-log file.

I/O is performed to the physical-log file in the following cases:

- KAIO is not implemented.
- The physical-log file is stored in buffered-file chunks.

Only when KAIO is implemented and the physical-log file is in raw disk space does the database server use a KAIO thread in the CPU virtual processor to perform I/O to the physical log.

The physical-log file stores *before-image*s of dbspace pages that have changed since the last *checkpoint*.

At the start of recovery, before processing transactions from the logical log, the database server uses the physical-log file to restore before-images to dbspace pages that have changed since the last checkpoint.

I/O to the physical-log file is the second-highest priority I/O after I/O to the logical-log files.

If the physical-log file is in a dbspace that is **not** mirrored, the database server runs only one PIO virtual processor.

If the physical-log file is in a dbspace that is mirrored, the database server runs two PIO virtual processors.

This class of virtual processors has no parameters associated with it.

Asynchronous I/O

The database server performs database I/O asynchronously, meaning that I/O is queued and performed independently of the thread that requests the I/O.

Performing I/O asynchronously allows the thread that makes the request to continue working while the I/O is being performed.

The database server performs all database I/O asynchronously, using one of the following facilities:

- AIO virtual processors
- KAIO on platforms that support it

Database I/O includes I/O for SQL statements, read-ahead, page cleaning, and checkpoints.

Kernel-asynchronous I/O

The database server implements KAIO by running a KAIO thread on the CPU virtual processor.

The KAIO thread performs I/O by making system calls to the operating system, which performs the I/O independently of the virtual processor.

The database server uses KAIO when the following conditions exist:

- The computer and operating system support it.
- A performance gain is realized.
- The I/O is to raw disk space.

The KAIO thread can produce better performance for disk I/O than the AIO virtual processor can, because it does not require a switch between the CPU and AIO virtual processors.

UNIX only: IBM® Informix® implements KAIO when Informix ports to a platform that supports this feature. The database server administrator does not configure KAIO. To see if KAIO is supported on your platform, see the machine notes file.

Linux only: Kernel asynchronous I/O (KAIO) is enabled by default. You can disable this by specifying that `KAIOOFF=1` in the environment of the process that starts the server.

On Linux®, there is a system-wide limit of the maximum number of parallel KAIO requests.

The `/proc/sys/fs/aio-max-nr` file contains this value.

The Linux system administrator can increase the value, for example, by using this command:

```
# echo new_value >
/proc/sys/fs/aio-max-nr
```

The current number of allocated requests of all operating system processes is visible in the `/proc/sys/fs/aio-nr` file.

By default, Dynamic Version allocates half of the maximum number of requests and assigns them equally to the number of configured CPU virtual processors.

You can use the environment variable KAIOON to control the number of requests allocated per CPU virtual processor.

Do this by setting KAIOON to the required value before starting Informix.

The minimum value for KAIOON is `100`.

If Linux is about to run out of KAIO resources, for example when dynamically adding many CPU virtual processors, warnings are printed in
the `online.log` file.

If this happens, the Linux system administrator must add KAIO resources as described previously.

AIO virtual processors

If the platform does not support KAIO or if the I/O is to buffered-file chunks, the database server performs database I/O through the AIO class of virtual processors.

All AIO virtual processors service all I/O requests equally within their class.

The database server assigns each disk chunk a queue, sometimes known as a *gfd queue*, which is based on the file name of the chunk.

The database server orders I/O requests within a queue according to an algorithm that minimizes disk-head movement.

The AIO virtual processors service queues that have pending work in round-robin fashion.

All other non-chunk I/O is queued in the AIO queue.

Use the VPCLASS parameter with the **`aio`** keyword to specify the number of AIO virtual processors that the database server starts initially.

You can start additional AIO virtual processors while the database server is in online mode.

You cannot drop AIO virtual processors while the database server is in online mode.

You can enable the database server to add AIO virtual processors and flusher threads when the server detects that AIO VPs are not keeping up with the I/O workload.

Include the **`autotune=1`** keyword in the VPCLASS configuration parameter setting.

Manually controlling the number of AIO VPs

The goal in allocating AIO virtual processors is to allocate enough of them so that the lengths of the I/O request queues are kept short; that is, the queues have as few I/O requests in them as possible.

When the *gfd* queues are consistently short, it indicates that I/O to the disk devices is being processed as fast as the requests occur.

The **`onstat-g ioq`** command shows the length and other statistics about I/O queues.

You can use this command to monitor the length of the *gfd* queues for the AIO virtual processors.

One AIO virtual processor might be sufficient:

- If the database server implements kernel asynchronous I/O (KAIO) on your platform and all of your dbspaces are composed of raw disk space
- If your file system supports direct I/O for the page size that is used for the dbspace chunk and you use direct I/O

Allocate two AIO virtual processors per active dbspace that is composed of buffered file chunks.

- If the database server implements KAIO, but you are using some buffered files for chunks
- IF KAIO is not supports by the system for chunks.

If KAIO is not implemented on your platform, allocate two AIO virtual processors for each disk that the database server accesses frequently.

If you use cooked files and if you enable direct I/O using the DIRECT_IO configuration parameter, you might be able to reduce the number of AIO virtual processors.

If the database server implements KAIO and you enabled direct I/O using the DIRECT_IO configuration parameter, IBM® Informix® attempts to use KAIO, so you probably do not require more than one AIO virtual processor.

However, even when direct I/O is enabled, if the file system does not support either direct I/O or KAIO, you still must allocate two AIO virtual processors for every active dbspace that is composed of buffered file chunks or does not use KAIO.

Temporary dbspaces do not use direct I/O. If you have temporary dbspaces, you probably require more than one AIO virtual processors.

Allocate enough AIO virtual processors to accommodate the peak number of I/O requests.

Generally, it is not detrimental to allocate too many AIO virtual processors.

Network virtual processors

A client can connect to the database server in the through following ways: a network connection, a pipe, or shared memory.

The network connection can be made by a client on a remote computer or by a client on the local computer mimicking a connection from a remote computer (called a *local-loopback connection*).

Specifying Network Connections

In general, the DBSERVERNAME and DBSERVERALIASES parameters define dbservernames that have corresponding entries in the `sqlhosts` file or registry.

Each dbservername parameter in `sqlhosts` has a **nettype** entry that specifies an interface/protocol combination.

The database server runs one or more poll threads for each unique **nettype** entry.

The NETTYPE configuration parameter provides optional configuration information for an interface/protocol combination.

You can use it to allocate more than one poll thread for an interface/protocol combination and also designate the virtual-processor class (CPU or NET) on which the poll threads run.

Run poll threads on CPU or network virtual processors

Poll threads can run either on CPU virtual processors or on network virtual processors.

In general, and particularly on a single-processor computer, poll threads run more efficiently on CPU virtual processors.

This might not be true, however, on a multiprocessor computer with many remote clients.

The NETTYPE parameter has an optional entry, called *vp class*, which you can use to specify either CPU or NET, for CPU or network virtual-processor classes, respectively.

If you do not specify a virtual processor class for the interface/protocol combination (poll threads) associated with the **`DBSERVERNAME`** variable, the class defaults to CPU.

The database server assumes that the interface/protocol combination associated with **`DBSERVERNAME`** is the primary interface/protocol combination and that it is the most efficient.

For other interface/protocol combinations, if no vp class is specified, the default is NET.

While the database server is in online mode, you cannot drop a CPU virtual processor that is running a poll or a listen thread.

Important: You must carefully distinguish between poll threads for network connections and poll threads for shared memory connections, which run one per CPU virtual processor. TCP connections must only be in network virtual processors, and you must only have the minimum required to maintain responsiveness. Shared memory connections must only be in CPU virtual processors and run in every CPU virtual processor.

Specify the number of networking virtual processors

Each poll thread requires a separate virtual processor, so you indirectly specify the number of networking virtual processors when you specify the number of poll threads for an interface/protocol combination and specify that they are to be run by the NET class.

If you specify CPU for the `vp class`, you must allocate a sufficient number of CPU virtual processors to run the poll threads.

If the database server does not have a CPU virtual processor to run a CPU poll thread, it starts a network virtual processor of the specified class to run it.

For most systems, one poll thread and consequently one virtual processor per network interface/protocol combination is sufficient.

For systems with 200 or more network users, running additional network virtual processors might improve throughput.

In this case, you must experiment to determine the optimal number of virtual processors for each interface/protocol combination.

Specify listen and poll threads for the client/server connection

When you start the database server, the `oninit` process starts an internal thread, called a *listen thread*, for each dbservername that you specify with the DBSERVERNAME and DBSERVERALIASES parameters in the `onconfig` file.

To specify a listen port for each of these dbservername entries, assign it a unique combination of **hostname** and **service name** entries in `sqlhosts`.

For example, the `sqlhosts` file or registry entry shown in the following table causes the database server **soc_ol1** to start a listen thread for **port1** on the host, or network address, **myhost**.

dbservername	nettype	hostname	service name
soc_ol1	Onsoctcp	myhost	port1

Table 1. A listen thread for each listen port

The listen thread opens the port and requests one of the poll threads for the specified interface/protocol combination to monitor the port for client requests.

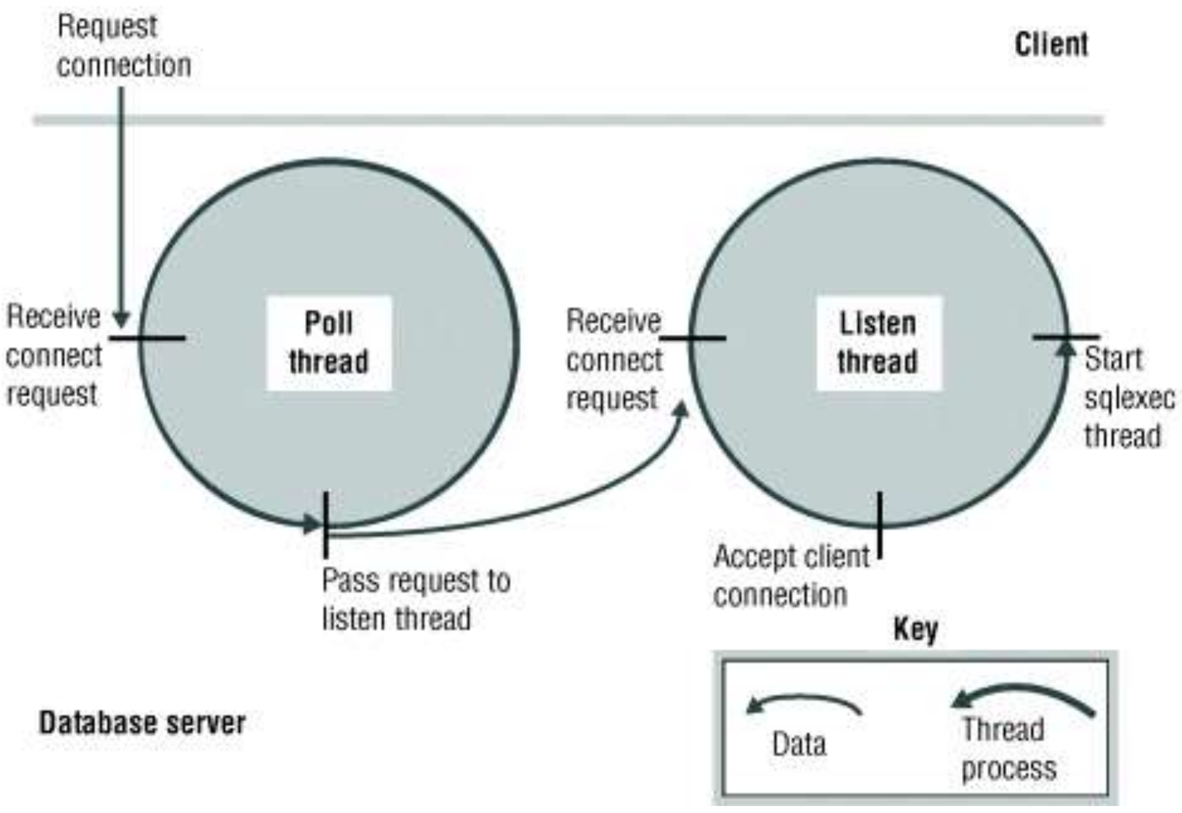
The poll thread runs either in the CPU virtual processor or in the network virtual processor for the connection that is being used.

When a poll thread receives a connection request from a client, it passes the request to the listen thread for the port.

The listen thread authenticates the user, establishes the connection to the database server, and starts an **sqlexec** thread, the session thread that performs the primary processing for the client.

The following figure illustrates the roles of the listen and poll threads in establishing a connection with a client application.

Figure 1. The roles of the poll and the listen threads in connecting to a client



A poll thread waits for requests from the client and places them in shared memory to be processed by the **sqlexec** thread.

For network connections, the poll thread places the message in a queue in the shared-memory global pool.

The poll thread then wakes up the **sqlexec** thread of the client to process the request.

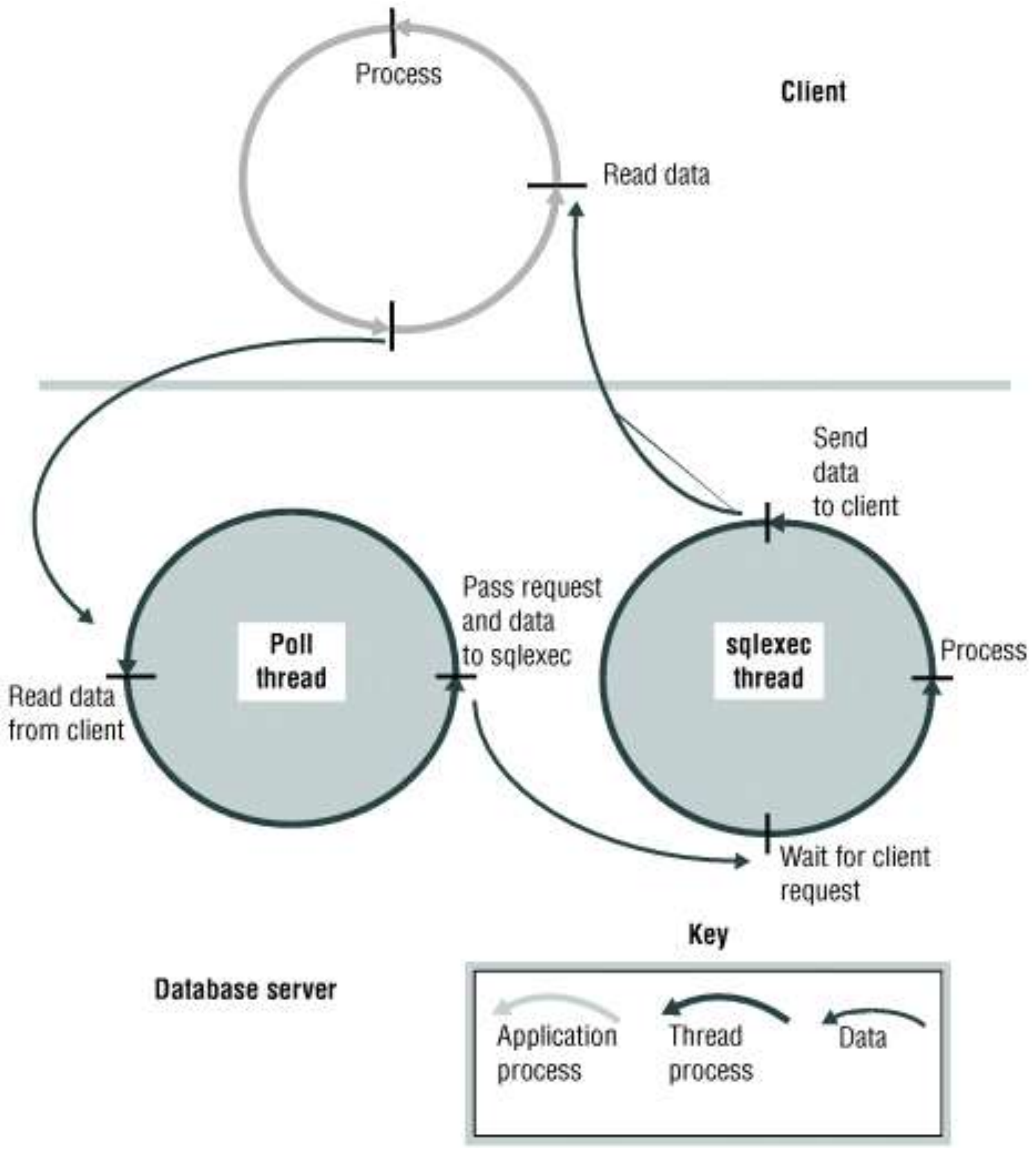
Whenever possible, the **sqlexec** thread writes directly back to the client without the help of the poll thread.

In general, the poll thread reads data from the client, and the **sqlexec** thread sends data to the client.

UNIX only: For a shared-memory connection, the poll thread places the message in the communications portion of shared memory.

The following figure illustrates the basic tasks that the poll thread and the **sqlexec** thread perform in communicating with a client application.

Figure 2. The roles of the poll and sqlexec threads in communicating with the client application



Fast polling

You can use the FASTPOLL configuration parameter to enable or disable fast polling of your network, if your operating-system platform supports fast polling.

Fast polling is beneficial if you have many connections. For example, if you have more than 300 concurrent connections with the database server, you can enable the FASTPOLL configuration parameter for better performance.

You can enable fast polling by setting the FASTPOLL configuration parameter to `1`.

If your operating system does not support fast polling, IBM® Informix® ignores the FASTPOLL configuration parameter.

Multiple listen threads

You can improve service for connection requests by using multiple listen threads.

If the database server cannot service connection requests satisfactorily for a given interface/protocol combination with a single port and corresponding listen thread, you can improve service for connection requests in the following ways:

- By adding listen threads for additional ports.
- By adding listen threads to the same port if you have the **onimcsoc** or **onsoctcp** protocol
- By adding another network-interface card.
- By dynamically starting, stopping, or restarting listen threads for a SOCTCP or TLITCP network protocol, using SQL administration API or **`onmode -P`** commands.

If you have multiple listen threads for one port for the **onsoctcp** protocol, the database server can accept new connections if a CPU VP connection is busy.

Add listen threads

When you start the database server, the **oninit** process starts a listen thread for servers with the server names and server alias names that you specify with the DBSERVERNAME and DBSERVERALIASES configuration parameters.

You can add listen threads for additional ports.

You can also set up multiple listen threads for one service (port) for the **onimcsoc** or **onsoctcp** protocol.

To add listen threads for additional ports, you must first use the DBSERVERALIASES parameter to specify dbservernames for each of the ports.

For example, the DBSERVERALIASES parameter in the following figure defines two additional dbservernames, **soc_ol2** and **soc_ol3**, for the database server instance identified as **soc_ol1**.

```
DBSERVERNAME        soc_ol1
DBSERVERALIASES
soc_ol2,soc_ol3
```

After you define additional dbservernames for the database server, you must specify an interface/protocol combination and port for each of them in the `sqlhosts` file or registry.

Each port is identified by a unique combination of **hostname** and **servicename** entries.

For example, the `sqlhosts` entries shown in the following table cause the database server to start three listen threads for the **onsoctcp** interface/protocol combination, one for each of the ports defined.

dbservername	nettype	hostname	service name
soc_ol1	onsoctcp	myhost	port1
soc_ol2	onsoctcp	myhost	port2
soc_ol3	onsoctcp	myhost	port3

Table 1. The sqlhosts entries to listen to multiple ports for a single interface/protocol combination

If you include a NETTYPE parameter for an interface/protocol combination, it applies to all the connections for that interface/protocol combination.

In other words, if a NETTYPE parameter exists for **onsoctcp** in the previous table, it applies to all of the connections shown.

In this example, the database server runs one *poll* thread for the **onsoctcp** interface/protocol combination unless the NETTYPE parameter specifies more.

Setting up multiple listen threads for one port for the onimcsoc or onsoctcp protocol

To set up multiple listen threads for one service (port) for the **onimcsoc** or **onsoctcp** protocol, specify DBSERVERNAME and DBSERVERALIASES information as follows:

- DBSERVERNAME *<name>*-*<n>*
- DBSERVERALIASES *<name1>*-*<n>*,*<name2>*

For example:

- To bring up two listen threads for the server with the DBSERVERNAME of `ifx`, specify:

```
DBSERVERNAME ifx-2
```

- To bring up two listen threads for DBSERVERALIASES `ifx_a` and `ifx_b`, specify:

```
DBSERVERALIASES ifx_a-2,ifx_b-2
```

Add a network-interface card

You can add a network-interface card to improve performance or connect the database server to multiple networks.

You might want to improve performance if the network-interface card for the host computer cannot service connection requests satisfactorily.

To support multiple network-interface cards, you must assign each card a unique **hostname** (network address) in `sqlhosts`.

For example, using the same dbservernames shown in Add listen threads, the `sqlhosts` file or registry entries shown in the following table cause the database server to start three listen threads for the same interface/protocol combination (as did the entries in Add listen threads).

In this case, however, two of the threads are listening to ports on one interface card (**myhost1**), and the third thread is listening to a port on the second interface card (**myhost2**).

dbservername	**nettype**	**hostname**	**service name**
soc_ol1	onsoctcp	myhost1	port1
soc_ol2	onsoctcp	myhost1	port2
soc_ol3	onsoctcp	myhost2	port1

Table 1. Example of sqlhosts entries to support two network-interface cards for the onsoctcp interface/protocol combination

Dynamically starting, stopping, or restarting a listen thread

You can dynamically start, stop, or stop and start a listen thread for a SOCTCP or TLITCP network protocol without interrupting existing connections.

For example, you might want to stop listen threads that are unresponsive and then start new ones in situations when other server functions are performing normally and you do not want to shut down the server.

The listen thread must be defined in the `sqlhosts` file for the server.

If necessary, before start, stop, or restart a listen thread, you can revise the `sqlhosts` entry.

To dynamically start, stop, or restart listen threads:

1. Run one of the following **`onmode -P`** commands:
   - **`onmode -P start`** *`server_name`*
   - **`onmode -P stop`** *`server_name`*
   - **`onmode -P restart`** *`server_name`*

2. Alternatively, if you are connected to the **sysadmin** database, either directly or remotely, you can run one of the following commands:
   - An **`admin()`** or **`task()`** command with the **`start listen`** argument, using the format
     ```
     EXECUTE FUNCTION
     task("start listen",
     "server_name");
     ```
   - An **`admin()`** or **`task()`** command with the **`stop listen`** argument, using the format
     ```
     EXECUTE FUNCTION
     task("stop listen"
     ,"server_name");
     ```
   - An **`admin()`** or **`task()`** command with the **`restart listen`** argument, using the format
     ```
     EXECUTE FUNCTION
     task("restart listen",
     "server_name");
     ```

For example, either of the following commands starts a new listen thread for a server named **ifx_serv2**:

```
onmode -P start ifx_serv2
EXECUTE FUNCTION task("start
listen", "ifx_serv2");
```

Communications support module virtual processor

The communications support module (CSM) class of virtual processors performs communications support service and communications support module functions.

The database server starts the same number of CSM virtual processors as the number of CPU virtual processors that it starts, unless the communications support module is set to GSSCSM to support single sign-on.

When the communications support module is GSSCSM, the database server starts only one CSM virtual processor.

Note: Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) instead.

Encrypt virtual processors

Use the VPCLASS configuration parameter with the `encrypt` keyword to configure encryption VPs.

If the `encrypt` option of the VPCLASS parameter is not defined in the `onconfig` configuration file, the database server starts one ENCRYPT VP the first time that any encryption or decryption functions defined for column-level encryption are called.

You can define multiple ENCRYPT VPs if necessary to decrease the time required to start the database server.

To add five ENCRYPT VPs, add information in the `onconfig` file as follows:

```
VPCLASS encrypt,num=5
```

You can modify the same information using the **`onmode`** utility, as follows:

```
onmode -p 5 encrypt
```

Audit virtual processor

The database server starts one virtual processor in the audit class (ADT) when you turn on audit mode by setting the ADTMODE parameter in the `onconfig` file to `1`.

Miscellaneous virtual processor

The miscellaneous virtual processor services requests for system calls that might require a very large stack, such as fetching information about the current user or the host-system name.

Only one thread runs on this virtual processor; it executes with a stack of 128 KB.

Basic text search virtual processors

A basic text search virtual processor is required to run basic text search queries.

A basic text search virtual processor is added automatically when you create a basic text search index.

A basic text search virtual processor runs without yielding; it processes one index operation at a time.

To run multiple basic text search index operations and queries simultaneously, create additional basic text search virtual processors.

Use the VPCLASS configuration parameter with the BTS keyword to configure basic text search virtual processors.

For example, to add five BTS virtual processors, add the following line to the `onconfig` and restart the database server:

```
VPCLASS bts,num=5
```

You can dynamically add BTS virtual processors by using the **`onmode -p`** command, for example:

```
onmode -p 5 bts
```

MQ messaging virtual processor

An MQ virtual processor is required to use MQ messaging.

When you perform MQ messaging transactions, an MQ virtual processor is created automatically.

An MQ virtual processor runs without yielding; it processes one operation at a time.

To perform multiple MQ messaging transactions simultaneously, create additional MQ virtual processors.

Use the VPCLASS configuration parameter with the MQ keyword to configure MQ virtual processors.

For example, to add five MQ virtual processors, add the following line to the `onconfig` and restart the database server:

```
VPCLASS mq,noyield,num=5
```

Web feature service virtual processor

A web feature service virtual processor is required to use web feature service for geospatial data.

When you run a WFS routine, a WFS virtual processor is created automatically.

A WFS virtual processor runs without yielding; it processes one operation at a time.

To run multiple WFS routines simultaneously, create additional WFS virtual processors.

Use the VPCLASS configuration parameter with the WFSVP keyword to configure WFS virtual processors.

For example, to add five WFS virtual processors, add the following line to the `onconfig` and restart the database server:

```
VPCLASS wfsvp,noyield,num=5
```

XML virtual processor

An XML virtual processor is required to perform XML publishing.

When you run an XML function, an XML virtual processor is created automatically.

An XML virtual processor runs one XML function at a time.

To run multiple XML functions simultaneously, create additional XML virtual processors.

Use the VPCLASS configuration parameter with the IDSXMLVP keyword to configure XML virtual processors.

For example, to add five XML virtual processors, add the following line to the `onconfig` and restart the database server:

```
VPCLASS idsxmlvp,num=5
```

You can dynamically add XML virtual processors by using the **`onmode -p`** command, for example:

```
onmode -p 5 idsxmlvp
```

Manage virtual processors

These topics describe how to set the configuration parameters that affect database server virtual processors, and how to start and stop virtual processors.

Set virtual-processor configuration parameters

Use the VPCLASS configuration parameter to designate a class of virtual processors (VPs), create a user-defined virtual processor, and specify options such as the number of VPs that the server starts, the maximum number of VPs allowed for the class, and the assignment of VPs to CPUs if processor affinity is available.

The table lists the configuration parameters that are used to configure virtual processors.

Parameter	Description
MULTIPROCESSOR	Set to `1` to support multiple CPU virtual processors, or to `0` for only a single CPU VP
NETTYPE	Specifies parameters for network protocol threads and virtual processors
SINGLE_CPU_VP	Set to `0` to enable user-defined CPU VPs, or to any other setting for only a single CPU VP
VPCLASS	Each defines a VP class and its properties, such as how many VPs of this class start when the server starts
VP_MEMORY_CACHE_KB	Speeds access to memory blocks by creating a private memory cache for each CPU virtual processor

Table 1. Configuration parameters for configuring virtual processors

Start and stop virtual processors

When you start the database server, the **`oninit`** utility starts the number and types of virtual processors that you specify directly and indirectly.

You configure virtual processors primarily through configuration parameters and, for network virtual processors, through parameters in the sqlhosts information.

You can use the database server to start a maximum of 1000 virtual processors.

After the database server is in online mode, you can start more virtual processors to improve performance, if necessary.

While the database server is in online mode, you can drop virtual processors of the CPU and user-defined classes.

To shut down the database server and stop all virtual processors, use the **`onmode -k`** command.

Add virtual processors in online mode

While the database server is running, you can start additional virtual processors for some virtual processor classes with the **`-p`** option of the **`onmode`** utility.

You can start these additional virtual processors with the **`-p`** option of the **`onmode`** utility.

You can add network virtual processors with the NETTYPE configuration parameter.

You can also start additional virtual processors for user-defined classes to run user-defined routines.

Add virtual processors in online mode with onmode

Use the **-p** option of the **`onmode`** command to add virtual processors while the database server is in online mode.

Specify the number of virtual processors that you want to add with a positive number.

As an option, you can precede the number of virtual processors with a plus sign (+).

Following the number, specify the virtual processor class in lowercase letters.

For example, either of the following commands starts four additional virtual processors in the AIO class:

```
onmode -p 4 aio

onmode -p +4 aio
```

The **`onmode`** utility starts the additional virtual processors immediately.

You can add virtual processors to only one class at a time.

To add virtual processors for another class, you must run **`onmode`** again.

Add network virtual processors

When you add network virtual processors, you add poll threads, each of which requires its own virtual processor to run.

In the following example, the poll threads handle a total of 240 connections:

```
NETTYPE ipcshm,4,60,CPU #
Configure poll thread(s) for
nettype
```

For ipcshm, the number of poll threads correspond to the number of memory segments.

For example, if NETTYPE is set to `3,100` and you want one poll thread, set the poll thread to `1,300`.

Drop CPU and user-defined virtual processors

While the database server is in online mode, you can use the **-p** option of the **onmode** utility to drop, or terminate, virtual processors of the CPU and user-defined classes.

Drop CPU virtual processors

Following the **onmode** command, specify a negative number that is the number of virtual processors that you want to drop, and then specify the CPU class in lowercase letters.

For example, the following command drops two CPU virtual processors:

```
% onmode -p -2 cpu
```

If you attempt to drop a CPU virtual processor that is running a poll thread, you receive the following message:

```
onmode: failed when trying to change
the number of cpu virtual processor
by -number.
```

Drop user-defined virtual processors

Following the **`onmode`** command, specify a negative number that is the number of virtual processors that you want to drop, and then specify the user-defined class in lowercase letters.

For example, the following command drops two virtual processors of the class *usr*:

```
onmode -p -2 usr
```

Windows only: In Windows, you can have only one user-defined virtual processor class at a time. Omit the *number* parameter in the **onmode -p** ***`vpclass`*** command.

Monitor virtual processors

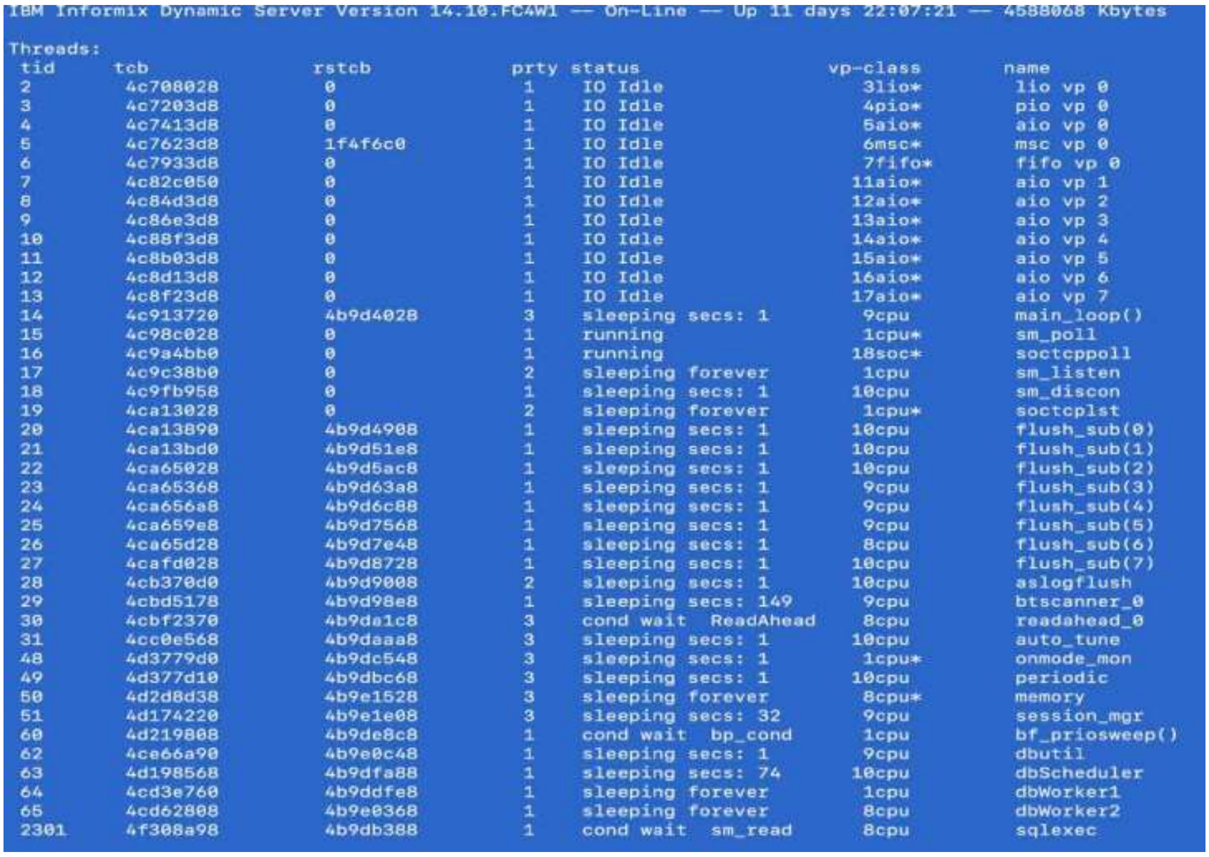
Monitor the virtual processors to determine if the number of virtual processors configured for the database server is optimal for the current level of activity.

Monitor virtual processors with command-line utilities

Use **`onstat -g`** options to monitor virtual processors.

The onstat -g ath command

The **onstat -g ath** command displays information about system threads and the virtual-processor classes.



```
IBM Informix Dynamic Server Version 14.10.FC4W1 -- On-Line -- Up 11 days 22:07:21 -- 4588068 Kbytes

Threads:
 tid      tcb             rstcb           prty status              vp-class      name
 2        4c708028        0               1    IO Idle              3lio*        lio vp 0
 3        4c7203d8        0               1    IO Idle              4pio*        pio vp 0
 4        4c7413d8        0               1    IO Idle              5aio*        aio vp 0
 5        4c7623d8        1f4f6c0         1    IO Idle              6msc*        msc vp 0
 6        4c7933d8        0               1    IO Idle              7fifo*       fifo vp 0
 7        4c82c050        0               1    IO Idle             11aio*        aio vp 1
 8        4c84d3d8        0               1    IO Idle             12aio*        aio vp 2
 9        4c86e3d8        0               1    IO Idle             13aio*        aio vp 3
 10       4c88f3d8        0               1    IO Idle             14aio*        aio vp 4
 11       4c8b03d8        0               1    IO Idle             15aio*        aio vp 5
 12       4c8d13d8        0               1    IO Idle             16aio*        aio vp 6
 13       4c8f23d8        0               1    IO Idle             17aio*        aio vp 7
 14       4c913720        4b9d4028        3    sleeping secs: 1     9cpu         main_loop()
 15       4c98c028        0               1    running              1cpu*        sm_poll
 16       4c9a4bb0        0               1    running             18soc*        soctcppoll
 17       4c9c38b0        0               2    sleeping forever     1cpu         sm_listen
 18       4c9fb958        0               1    sleeping secs: 1    10cpu         sm_discon
 19       4ca13028        0               2    sleeping forever     1cpu*        soctcplst
 20       4ca13890        4b9d4908        1    sleeping secs: 1    10cpu         flush_sub(0)
 21       4ca13bd0        4b9d51e8        1    sleeping secs: 1    10cpu         flush_sub(1)
 22       4ca65028        4b9d5ac8        1    sleeping secs: 1    10cpu         flush_sub(2)
 23       4ca65368        4b9d63a8        1    sleeping secs: 1     9cpu         flush_sub(3)
 24       4ca656a8        4b9d6c88        1    sleeping secs: 1     9cpu         flush_sub(4)
 25       4ca659e8        4b9d7568        1    sleeping secs: 1     9cpu         flush_sub(5)
 26       4ca65d28        4b9d7e48        1    sleeping secs: 1     8cpu         flush_sub(6)
 27       4cafd028        4b9d8728        1    sleeping secs: 1    10cpu         flush_sub(7)
 28       4cb370d0        4b9d9008        2    sleeping secs: 1    10cpu         aslogflush
 29       4cbd5178        4b9d98e8        1    sleeping secs: 149   9cpu         btscanner_0
 30       4cbf2370        4b9da1c8        3    cond wait  ReadAhead 8cpu         readahead_0
 31       4cc0e568        4b9daaa8        3    sleeping secs: 1    10cpu         auto_tune
 48       4d3779d0        4b9dc548        3    sleeping secs: 1     1cpu*        onmode_mon
 49       4d377d10        4b9dbc68        3    sleeping secs: 1    10cpu         periodic
 50       4d2d8d38        4b9e1528        3    sleeping forever     8cpu*        memory
 51       4d174220        4b9e1e08        3    sleeping secs: 32    9cpu         session_mgr
 60       4d219808        4b9de8c8        1    cond wait  bp_cond   1cpu         bf_priosweep()
 62       4ce66a90        4b9e0c48        1    sleeping secs: 1     9cpu         dbutil
 63       4d198568        4b9dfa88        1    sleeping secs: 74   10cpu         dbScheduler
 64       4cd3e760        4b9ddfe8        1    sleeping forever     1cpu         dbWorker1
 65       4cd62808        4b9e0368        1    sleeping forever     8cpu         dbWorker2
 2301     4f308a98        4b9db388        1    cond wait  sm_read   8cpu         sqlexec
```

The onstat -g glo command

Use the **`onstat -g glo`** command to display information about each virtual processor that is currently running, and cumulative statistics for each virtual processor class.

The onstat -g ioq command

Use the **`onstat -g ioq`** option to determine whether you must allocate additional virtual processors.

The command **`onstat -g ioq`** displays the length and other statistics about I/O queues.

If the length of the I/O queue is growing, I/O requests are accumulating faster than the AIO virtual processors can process them.

If the length of the I/O queue continues to show that I/O requests are accumulating, consider adding AIO virtual processors.

The onstat -g rea command

Use the **`onstat -g rea`** option to monitor the number of threads in the ready queue.

If the number of threads in the ready queue is growing for a class of virtual processors (for example, the CPU class), you might be required to add more virtual processors to your configuration.

Monitor virtual processors with SMI tables

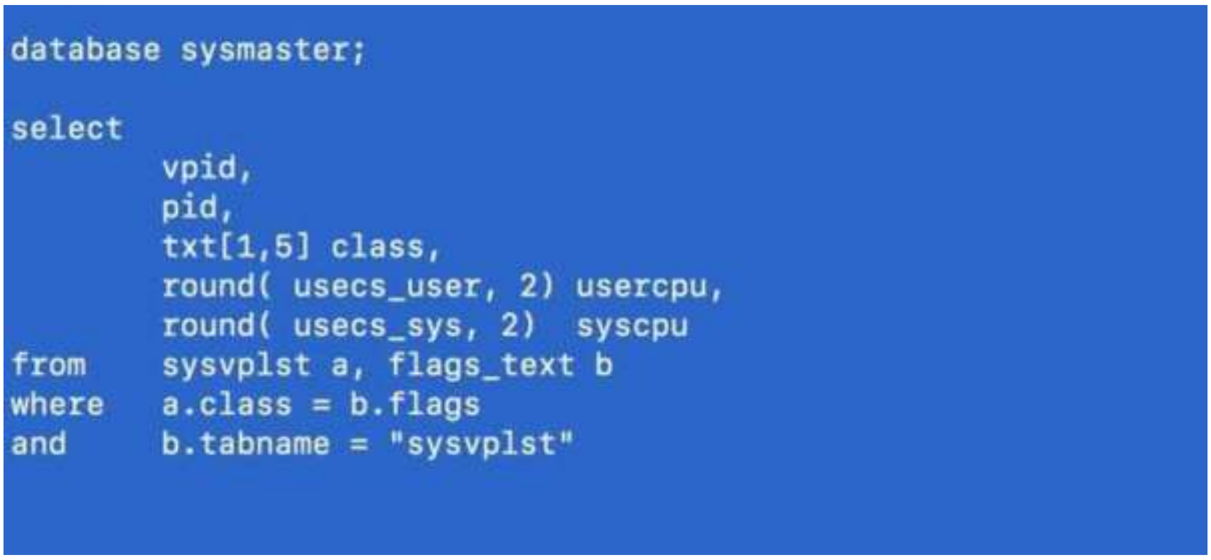
Query the **sysvpprof** table to obtain information about the virtual processors that are currently running.

This table contains the following columns.

Column	Description
vpid	Virtual-processor ID number
class	Virtual-processor class
usercpu	Minutes of user CPU used
syscpu	Minutes of system CPU used

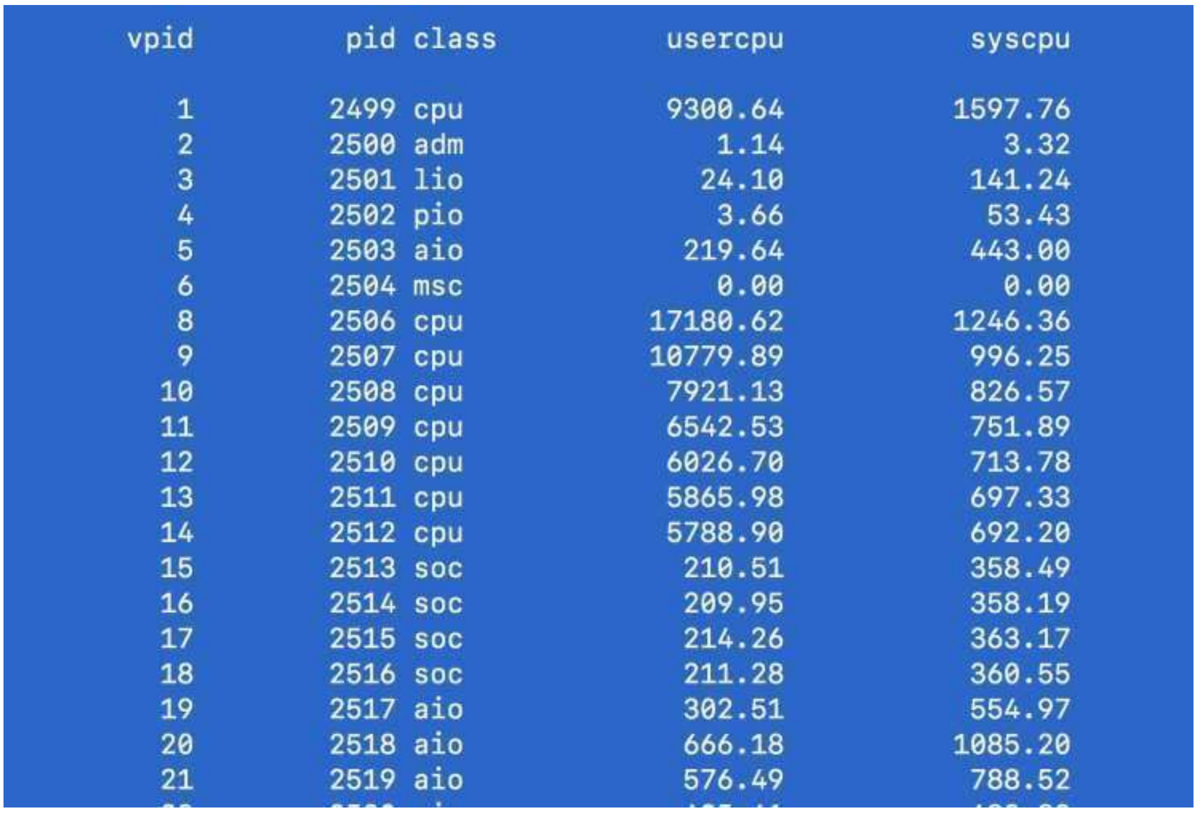
Oninit CPU Usage

vpprof.sql



```
database sysmaster;

select
        vpid,
        pid,
        txt[1,5] class,
        round( usecs_user, 2) usercpu,
        round( usecs_sys, 2)  syscpu
from    sysvplst a, flags_text b
where   a.class = b.flags
and     b.tabname = "sysvplst"
```



```
      vpid          pid class          usercpu          syscpu

         1         2499 cpu            9300.64         1597.76
         2         2500 adm               1.14            3.32
         3         2501 lio              24.10          141.24
         4         2502 pio               3.66           53.43
         5         2503 aio             219.64          443.00
         6         2504 msc               0.00            0.00
         8         2506 cpu           17180.62         1246.36
         9         2507 cpu           10779.89          996.25
        10         2508 cpu            7921.13          826.57
        11         2509 cpu            6542.53          751.89
        12         2510 cpu            6026.70          713.78
        13         2511 cpu            5865.98          697.33
        14         2512 cpu            5788.90          692.20
        15         2513 soc             210.51          358.49
        16         2514 soc             209.95          358.19
        17         2515 soc             214.26          363.17
        18         2516 soc             211.28          360.55
        19         2517 aio             302.51          554.97
        20         2518 aio             666.18         1085.20
        21         2519 aio             576.49          788.52
```